iRMX 86™
PROGRAMMING TECHNIQUES

Manual Order Number: 142982-001

PUBLICATION NOTICE

This document is the First Edition of the iRMX 86™ PROGRAMMING TECHNIQUES MANUAL. This manual reflects the software associated with Release 2.0 of the iRMX 86 Operating System. This edition applies to all future software releases until further notice.

PRINTING HISTORY:

Manual Edition:	Software Release:	Print Date:
First	2.0	November, 1980

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

A275/1180/3K DD

PREFACE

The techniques presented in this manual relate to one or more of the following phases of system development:

- Dividing an application into jobs and tasks.
- Writing the code for tasks.
- Configuring and starting up the system.
- Debugging an application.

Because of the time and effort that these techniques can save, you should refer to this manual as you enter each of these phases in the development of your system.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

TABLES

CHAPTER 1. ORGANIZATION OF THIS MANUAL

This manual provides a number of techniques to reduce the amount of time and effort you must spend designing and implementing your iRMX 86-based application system. Each chapter either provides background information or directly attacks a specific type of problem.

CHAPTER OUTLINE

In order to help you quickly decide if a particular technique is of use to you, all the chapters (with the exception of this one) are organized as follows:

1) Assumptions about the Reader

   This section describes the readers that are likely to be interested in the content of the chapter. This section also specifies any prerequisite knowledge that readers must possess in order to thoroughly understand the chapter.

2) Intent of the Chapter

   This section briefly explains how the contents of the chapter might apply to your situation. After reading only one or two paragraphs, you will be able to decide whether to continue reading.

3) Technique or Explanation

   This part of the chapter, which may consist of one or more sections, provides the information promised as being the purpose of the chapter.

4) Additional Reading

   This section contains a list of reading material that provides the prerequisite knowledge required for the chapter. Chapters requiring no prerequisite knowledge have no bibliography.

INDEXES

This manual is equipped with two indexes. The first one, Index A, lists the chapters according to when they might be useful to you. For instance, chapters useful during debugging are listed in one place, while chapters useful during configuration are listed in another. Whenever you move from one phase of system development to another, you should consult this index.

The second index, Index B, provides the conventional alphabetical listing of topics and the pages on which they are discussed or significantly mentioned.

CHAPTER 2. SELECTING A PL/M-86 SIZE CONTROL

This chapter applies to you only if you have decided to program your iRMX 86 tasks using PL/M-86. In order to understand the following explanation, you should be familiar with

- The PL/M-86 programming language
- PL/M-86 models of computation
- iRMX 86 jobs, tasks, and segments

If you are unfamiliar with any of these items, refer to the bibliography at the end of this chapter for titles of manuals that can provide background information.

PURPOSE OF THIS CHAPTER

Whenever you invoke the PL/M-86 Compiler, you must specify (either explicitly or by default) a program size control (SMALL, COMPACT, MEDIUM, or LARGE). This size control determines which model of computation the compiler uses and, consequently, greatly affects the amount of memory required to store your application's object code.

The following section explains which size control to use in order to produce the smallest object program while still satisfying the requirements of your system.

MAKING THE SELECTION

When you compile your programs using the PL/M-86 SMALL control, all POINTER values are 16 bits long. This leads to a number of restrictions, including the inability to address the contents of an iRMX 86 segment that has been received from another job. Because of these restrictions, the iRMX 86 Operating System is currently not compatible with PL/M-86 procedures compiled using the SMALL size control.

Since you cannot use the SMALL size control, you must choose between COMPACT, MEDIUM and LARGE. The algorithm for selecting a size control is presented later in this chapter. However, before you examine the algorithm, you should be aware that your choice can place restrictions on your system.

RAMIFICATIONS OF YOUR SELECTION

If you decide to use the COMPACT or MEDIUM size controls, the capabilities of your system will be slightly restricted. Only the LARGE size control preserves all of the features of the system.

Restrictions Associated with Compact

If you decide to use PL/M-86 COMPACT, you will not be able to use exception handlers. However, you can still process exceptional conditions by dealing with them in your task's code.

Restrictions Associated with Medium

If you decide to use PL/M-86 MEDIUM, you lose the option of having the iRMX 86 Operating System dynamically allocate stacks for tasks that are created dynamically. This means that you must anticipate the stack requirements of each such task, and you must explicitly reserve memory for each stack during the process of configuring the system.

DECISION ALOGRITHM

Before you attempt to use the flowchart (Figure 2-1) to make your decision, note that three of the boxes are numbered. Each of these three boxes asks you to derive a quantity that represents a memory requirement of your iRMX 86 job. In order to derive the quantity requested in each of the boxes, follow the directions provided below in the section having the same number as the box.

1. COMPUTE MEMORY REQUIREMENTS FOR STATIC DATA

   Box 1 asks for an estimate of the amount of memory required to store the static data for all the tasks of your iRMX 86 job. Static data consists of all variables other than:

   - parameters in a procedure call
   - variables local to a reentrant PL/M-86 procedure
   - PL/M-86 structures that are declared to be BASED

   To obtain an accurate estimate of this quantity, use the COMPACT size control to compile the code for each task in your job. For each compilation, find the MODULE INFORMATION area at the end of the listing. Within this area is a quantity labeled VARIABLE AREA SIZE and another labeled CONSTANT AREA SIZE.

Now you must compute the static data size for each individual compilation by adding the VARIABLE AREA SIZE to the CONSTANT AREA SIZE.

Once you have computed the static data size for each compilation in the job, add them to obtain the static data size for the entire job.

2. COMPUTE MEMORY REQUIREMENTS FOR CODE

Box 2 asks for an estimate of the amount of memory required to store the code for all the tasks of your iRMX 86 job. To obtain this estimate, perform the following steps:

- Using the COMPACT size control, compile the code for each task in your job.

- For each compilation, find the MODULE INFORMATION area at the end of the listing. In this area is a value labeled CODE AREA SIZE. This value is the amount of memory required to store the code generated by this individual compilation.

- Sum the code requirements for all the compilations in the job. The result is the code requirement for the entire job.

3. COMPUTE MEMORY REQUIREMENTS FOR STACK

Box 3 asks for an estimate of the amount of memory required to store the stacks of all the tasks in your iRMX 86 job. If you plan to have the iRMX 86 Operating System create your stacks dynamically, your stack requirement (for the purpose of the flowchart) is zero.

If, on the other hand, you plan to create the stacks yourself, you can estimate the memory requirements by performing the following steps. Refer to the MODULE INFORMATION AREA of the compilation listings that you obtained while working with Box 2. Within this area is a value labeled MAXIMUM STACK SIZE. To this number, add the system stack requirement that you can determine by following the procedure in Chapter 11 of this manual. The result is an estimate of the stack requirement for one compilation. To compute the requirements for the entire job, just sum the requirements for all the compilations in the job.

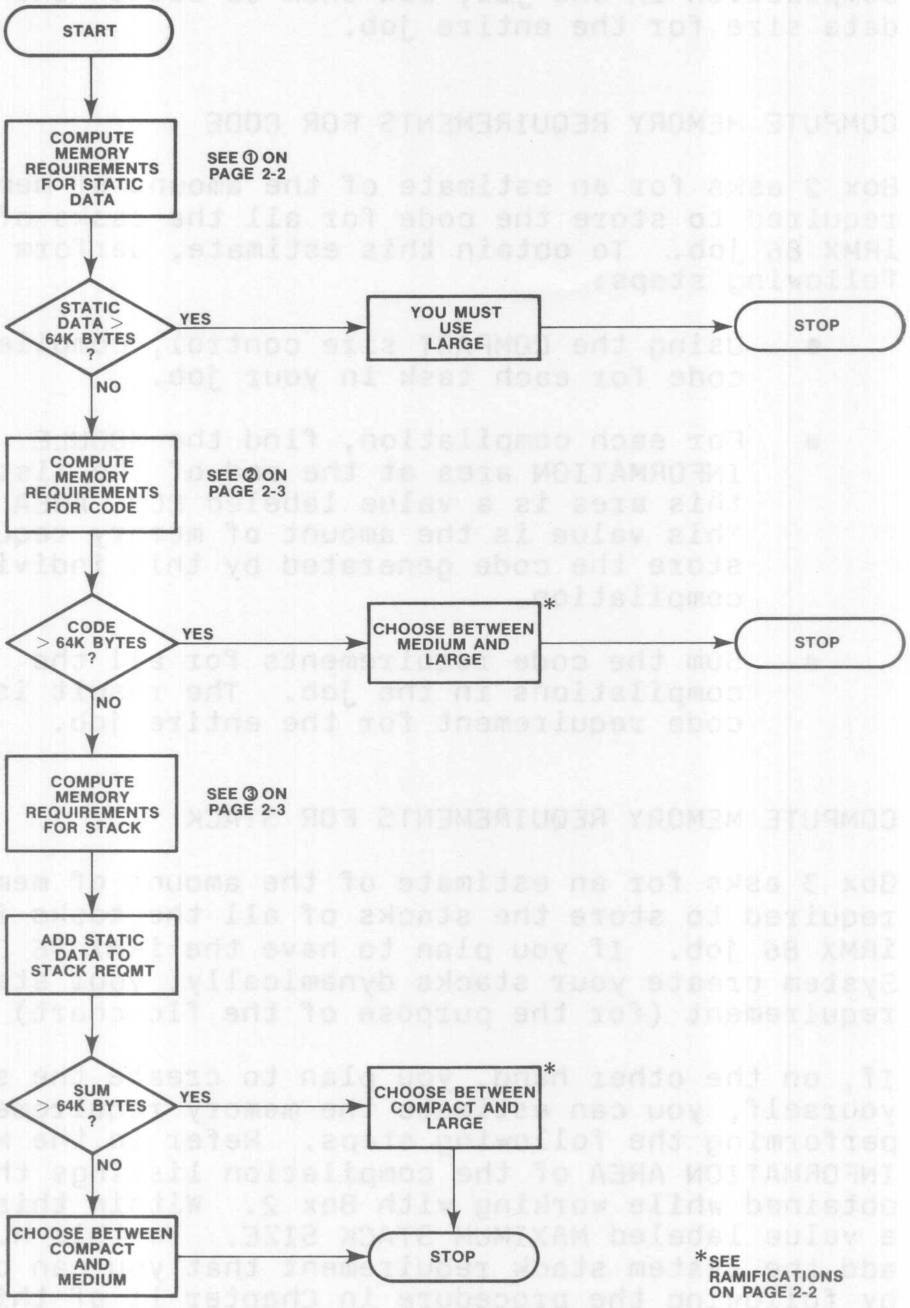

Figure 2-1. Decision Algorithm for Size Control

BIBLIOGRAPHY

The following literature contains information that you might need to be acquainted with before you select a PL/M-86 size control:

- PL/M-86 PROGRAMMING MANUAL
  Order Number 9800466

  This manual describes the PL/M-86 language.

- ISIS-II PL/M-86 COMPILER OPERATOR'S MANUAL
  Order Number 9800478

  This manual describes the differences between the various PL/M-86 size controls (or models of computation).

- iRMX 86™ NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL
  Order Number 9803122

  This manual contains detailed descriptions of iRMX 86 segments, jobs and tasks. It also explains how you can tell the iRMX 86 System to create a task's stack dynamically.

- Chapter 11 of this manual

  Chapter 11 explains how to compute the amount of stack that the iRMX 86 Operating System requires.

CHAPTER 3. INTERFACE PROCEDURES AND LIBRARIES

This chapter is for anyone who writes programs that use iRMX 86 system calls. In order to understand this chapter, you should be familiar with the following concepts:

- the notion of system call
- the process of linking object modules
- the notion of an object library
- PL/M-86 size control

If you are unfamiliar with any of these concepts, refer to the bibliography at the end of this chapter for additional reading.

PURPOSE OF THIS CHAPTER

Familiarity with interface procedures is a prerequisite to understanding several of the programming techniques discussed later in this manual. The primary purpose of this chapter is to define the concept of an interface procedure and explain how it is used in the iRMX 86 Operating System.

DEFINITION OF INTERFACE PROCEDURE

The iRMX 86 Operating System uses interface procedures to simplify the process of calling one software module from another. In order to illustrate the usefulness of interface procedures, let's examine what happens without them.

Suppose you are writing an application task that will run in some hypothetical operating system. Figure 3-1 shows your application task calling two system procedures. If the system calls are direct (without an interface procedure serving as an intermediary), the application task must be bound to the system procedures either during compilation or during linking. Such binding causes your application task to be dependent upon the memory location of the system procedures.

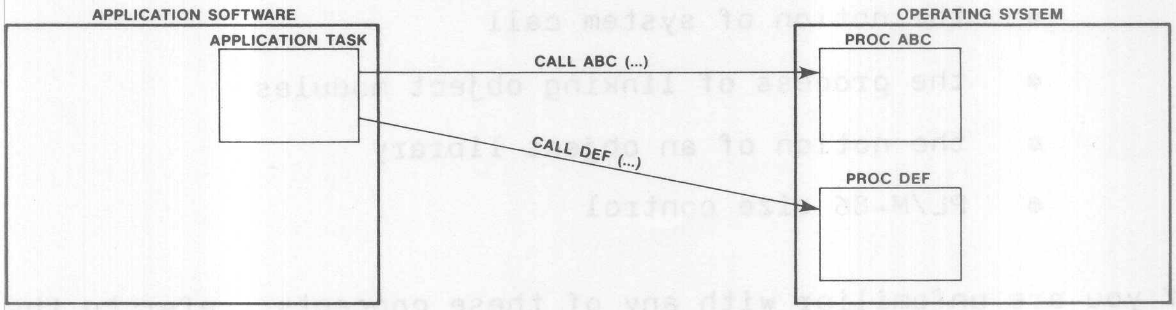

Figure 3-1. Direct location-dependent invocation

Now suppose that someone updates your operating system. If, during the process of updating the system, some of the system procedures are moved to different memory locations, then your application software must be relinked to the new operating system.

There are techniques for calling system procedures that do not assume unchanging memory locations. However, most of these techniques are complex (Figure 3-2) and assume that the application programmer is intimately familiar with the interrupt architecture of the processor.

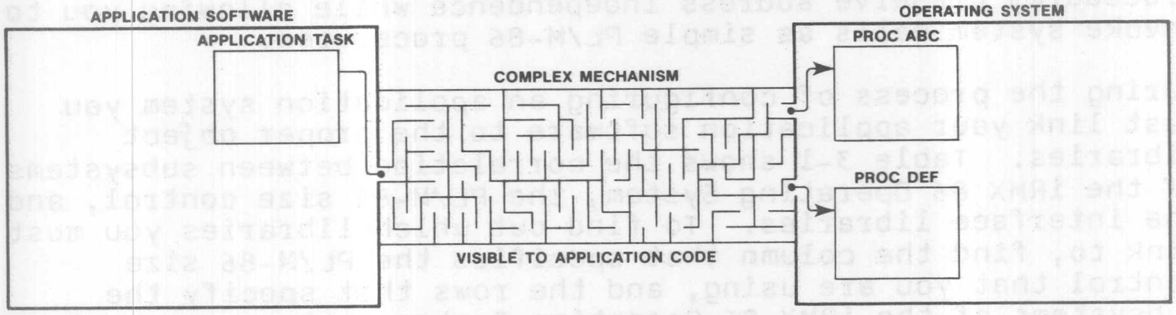

Figure 3-2. Complex location-independent invocation

The iRMX 86 Operating System uses interface procedures to mask the details of location-independent invocation from the application software (Figure 3-3). Whenever application programmers need to call a system procedure from application code, they use a simple procedure call (known as a system call). This system call invokes an interface procedure which, in turn, invokes the actual system procedure.

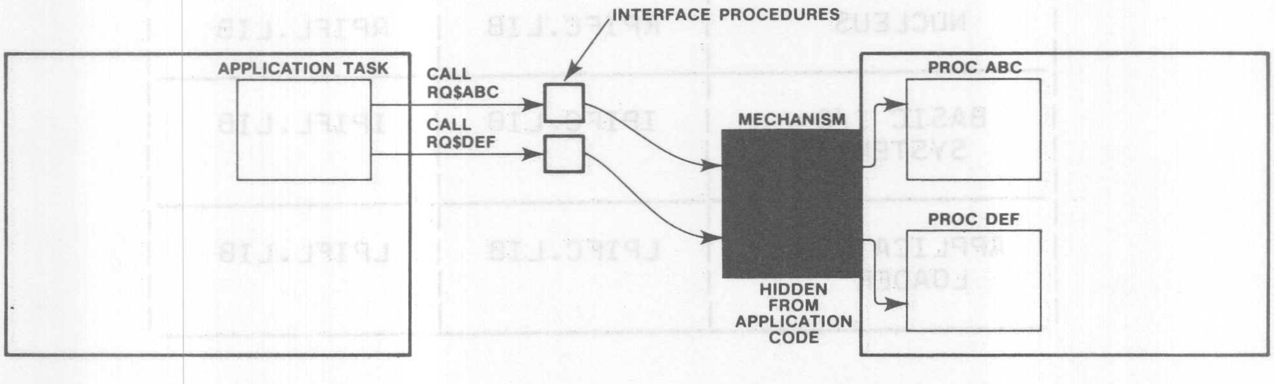

Figure 3-3. Simple invocation using an interface procedure

INTERFACE LIBRARIES

The iRMX 86 Operating System provides you with a set of object code libraries containing PL/M-86 interface procedures. These procedures preserve address independence while allowing you to invoke system calls as simple PL/M-86 procedures.

During the process of configuring an application system you must link your application software to the proper object libraries. Table 3-1 shows the correlation between subsystems of the iRMX 86 Operating System, the PL/M-86 size control, and the interface libraries. To find out which libraries you must link to, find the column that specifies the PL/M-86 size control that you are using, and the rows that specify the subsystems of the iRMX 86 Operating System that you are using. You must link to the libraries that are named at the intersections of the column and the rows.

TABLE 3-1
INTERFACE LIBRARIES AS A FUNCTION OF
PL/M-86 SIZE CONTROL AND iRMX 86 SUBSYSTEMS

	COMPACT	LARGE OR MEDIUM
NUCLEUS	RPIFC.LIB	RPIFL.LIB
BASIC I/O SYSTEM	IPIFC.LIB	IPIFL.LIB
APPLICATION LOADER	LPIFC.LIB	LPIFL.LIB

BIBLIOGRAPHY

The following reading material contains information that relates to interface procedures and libraries.

- Chapter 2 of iRMX 86™ PROGRAMMING TECHNIQUES

  This chapter contains an algorithm for selecting a PL/M-86 size control.

- INTRODUCTION TO THE iRMX 86™ OPERATING SYSTEM
  Order Number 9803124

  This manual provides a general discussion of system calls.

- iRMX 86™ CONFIGURATION GUIDE FOR ISIS-II USERS
  Order Number 9803126

  This manual discusses the entire process of configuring an iRMX 86-based application system, including the details about linking interface libraries to application systems.

- MCS-86 SOFTWARE DEVELOPMENT UTILITIES OPERATING INSTRUCTIONS FOR ISIS-II USERS
  Order Number 9800639

  This manual describes the process of using libraries and the process of linking software modules.

- iRMX 86™ SYSTEM PROGRAMMER'S REFERENCE MANUAL
  Order Number 142721

  This manual describes interface procedures in more detail.

CHAPTER 4. EDITING INCLUDE FILES

This chapter is for anyone who writes PL/M-86 programs that use iRMX 86 system calls. In order to understand this chapter, you should be familiar with the following concepts:

- system calls
- INCLUDE files
- external procedures
- system configuration
- text editors

If you are unfamiliar with any of these concepts, refer to the bibliography at the end of this chapter for additional reading.

PURPOSE OF THIS CHAPTER

You received, as part of the iRMX 86 product, three INCLUDE files (Table 4-1). Each of these files is associated with one iRMX 86 subsystem, and each file contains the PL/M-86 external procedure declarations for all of the system calls provided by the associated subsystem.

TABLE 4-1

CORRELATION BETWEEN iRMX 86™ SUBSYSTEMS AND INCLUDE FILES THAT DECLARE SYSTEM CALLS

NAME OF SUBSYSTEM	NAME OF INCLUDE FILE
NUCLEUS	NUCLUS.EXT
BASIC I/O SYSTEM	IOS.EXT
APPLICATION LOADER	LOADER.EXT

EDITING INCLUDE FILES

When writing application code in PL/M-86, you must add to your source code an external procedure declaration for each iRMX 86 system call used by your code. The INCLUDE files provide a means of incorporating all of the system calls associated with any particular subsystem of the Operating System. However, if you do include all of the external procedure declarations (rather than including only those that your code actually uses), you may cause the PL/M-86 Compiler to run out of dynamic memory. This, in turn, will cause the compilation to terminate unsuccessfully.

This chapter tells you how to avoid overloading the PL/M-86 Compiler.

BEING SELECTIVE

You can reduce the probability of using up all of the Compiler's dynamic memory if you are selective about which declarations you incorporate into your source code. For instance, if the code you are compiling does not use the SEND$UNITS system call, you need not incorporate into your source the declaration of the RQ$SEND$UNITS procedure.

Since the INCLUDE files supplied by INTEL contain all of the declarations for each iRMX 86 subsystem, you cannot INCLUDE these files and be selective. In order to allow selective inclusion, you must divide the provided INCLUDE files into customized files. There are two methods you should consider.

THE SINGLETON METHOD

You can use a text editor (CREDIT for example) to divide each of the large INCLUDE files into many small INCLUDE files, each of which contains only one procedure declaration. Then, when you write application code, you selectively INCLUDE only the files that contain procedure declarations required by the code.

The advantage of this technique is that once all the small files have been created, no programmers need ever perform the division process again. The disadvantages are that someone must initially divide the large INCLUDE files into smaller files and that more INCLUDE statements are required.

THE SPECIAL-CASE METHOD

An alternative to the singleton method is to create one INCLUDE file that contains only the system calls required by the code being written. This method has the advantage of requiring less time initially. However, if at some later time you write more code that uses different system calls, you must invest additional time to create another special-case INCLUDE file.

TWO OTHER REASONS FOR EDITING INCLUDE FILES

There are two other reasons why you may want to edit INCLUDE files.

FINDING ERRORS DURING COMPILATION

If you edit your INCLUDE files to support only the system calls configured into your system, you can detect during compilation some errors that would otherwise appear only at runtime. Consider the following hypothetical example.

Suppose that your system does not use semaphores and that your system's configuration does not include any system call associated with semaphores. Also suppose that one of your programmers is unaware of this restriction and attempts to create and use semaphores. If external procedure declarations for semaphore-related system calls have not been INLCUDED in your source code, this oversight will be detected (as references to undefined symbols) when the source code is compiled. If, on the other hand, the declarations have been INCLUDEd, the error will not be detected until the system is run and an E$NOTCONFIGURED exceptional condition is generated.

ENFORCING THE DISTINCTION BETWEEN SYSTEM AND APPLICATION PROGRAMMERS

By editing the INCLUDE files, you can enforce the distinction between system programmers and application programmers. To accomplish this, create two collections of procedure declarations -- one for each type of programmer. Then, if application programmers attempt to use restricted system calls, the compiler will mark the illicit system calls as references to undefined symbols.

BIBLIOGRAPHY

The following reading material contains information that relates to INCLUDE files, iRMX 86 system calls, PL/M-86 external procedures, system configuration, and text editors:

- INTRODUCTION TO THE iRMX 86™ OPERATING SYSTEM
  Order Number 9803124

  This manual provides a general discussion of iRMX 86 system calls.

- ISIS-II PL/M-86 COMPILER OPERATOR'S MANUAL
  Order Number 9800478

  This manual discusses the concept of using INCLUDE files to add source statements to a file being compiled.

- PL/M-86 PROGRAMMING MANUAL
  Order Number 9800466

  This manual describes the PL/M-86 language and contains an explanation of external procedures.

- iRMX 86™ CONFIGURATION GUIDE FOR ISIS-II USERS
  Order Number 9803126

  This manual discusses the process of configuring an iRMX 86-based system.

- ISIS-II CREDIT (CRT-BASED TEXT EDITOR) USER'S GUIDE
  Order Number 9800902

  This manual explains how to use the CREDIT text editor.

CHAPTER 5. TIMER ROUTINES

This chapter is for anyone who writes programs that must determine approximate elapsed time. In order to make use of this chapter, you should be familiar with the following concepts:

- INCLUDE files
- iRMX 86 interface procedures
- iRMX 86 tasks
- initialization tasks
- using the LINK86 command of the MCS-86 Software Development Utilities

Furthermore, if you want to understand how the timer routines work, you must be fluent in PL/M-86 and know how to use iRMX 86 regions. The bibliography at the end of this chapter refers to text that discusses these topics.

PURPOSE OF THIS CHAPTER

The iRMX 86 Basic I/O System provides GET$TIME and SET$TIME system calls. These two calls supply your application with a timer having units of one second. However, if your application requires no features of the I/O System other than the timer, you can reduce your memory requirements by dropping the I/O System altogether and implementing the timer in your application.

This chapter provides the source code needed to build a timer into your application. These routines have been compiled and tested.

PROCEDURES IMPLEMENTING THE TIMER

Four PL/M-86 procedures are used to implement the timer. In brief, the procedures are:

- get_time

  This procedure requires no input parameter and returns a double word (POINTER) value equal to the current contents of the timer in seconds. This procedure can be called any number of times.

- set_time

  This procedure requires a double word (POINTER) input parameter that specifies the value (in seconds) to which you want the timer set. This procedure can be called any number of times.

- init_time

  This procedure creates the timer, initializes it to zero seconds, and starts it running. This procedure requires as input a POINTER to the WORD which is to receive the status of the initialization. This status will be zero if the timer is successfully created and nonzero otherwise. This procedure should be called only once.

- maintain_time

  This procedure is not called directly by your application. Rather, it runs as an iRMX 86 task that is created when your application calls init_time. The purpose of this task is to increment the contents of the timer once every second.

RESTRICTIONS

There are two important restrictions that you should keep in mind when using the timer routines:

CALL init_time FIRST

Before calling set_time or get_time, your application must call init_time. You can accomplish this by calling the init_time procedure from your job's initialization task.

ONLY ONE TIMER

These procedures implement only one timer. They do not allow you to maintain a different timer for each of several purposes. For example, if one job changes the contents of the timer (by using the set_time procedure), all jobs accessing the timer will be affected.

SOURCE CODE

You can compile the following PL/M-86 source code as a single module. This will yield an object module that you can link to your application code. However, before compiling these procedures, you must create files containing the external procedure declarations for the iRMX 86 interface procedures. The names of these files are specified in the $INCLUDE statements below.

```
$title('INDEPENDENT TIMER PROCEDURES')
/*****************************************************************
*                                                                *
*   This module consists of four procedures which implement     *
*   a timer having one-second granularity.  The outside world   *
*   has access to only three of these procedures-               *
*                                                                *
*       init_time                                                *
*       set_time                                                 *
*       get_time                                                 *
*                                                                *
*   The fourth procedure, maintain_time, is invoked              *
*   by init_time and is run as an iRMX 86 task to                *
*   measure time and increment the time counter.                 *
*****************************************************************/

timer:  DO;

/*****************************************************************
*   The following LITERALLY statements are used to improve      *
*   the readability of the code.                                 *
*****************************************************************/

    DECLARE
        FOREVER             LITERALLY 'WHILE OFFH',
        DWORD               LITERALLY 'POINTER',
        TOKEN               LITERALLY 'WORD',
        REGION              LITERALLY 'TOKEN',
        E$OK                LITERALLY '00000H',
        PRIORITY_QUEUE      LITERALLY '1',
        TASK                LITERALLY 'TOKEN';
```

```
/****************************************************************
*                                                               *
*   The following INCLUDE statements cause the external         *
*   procedure declarations for some of the iRMX 86              *
*   system calls to be included in the source code.             *
*                                                               *
****************************************************************/

$INCLUDE(:fl:icrtas.ext)  /* rq$create$task interface proc.*/
$INCLUDE(:fl:icrreg.ext)  /* rq$create$region   "        " */
$INCLUDE(:fl:isleep.ext)  /* rq$sleep           "        " */
$INCLUDE(:fl:idereg.ext)  /* rq$delete$region   "        " */
$INCLUDE(:fl:iregio.ext)  /* rq$send$control    "        " */
                          /* and  rq$receive$control "    " */


$subtitle('Local Data')
/****************************************************************
*   The following variables can be accessed by all of the       *
*   procedures in this module.                                  *
****************************************************************/

        DECLARE
            time_region         REGION,       /* Guards access to
                                                 time_in_sec.*/

            time_in_sec         DWORD,        /* Contains time in
                                                 seconds.*/

                                              /* Overlay        */
            time-in_sec_o       STRUCTURE(    /* used to obtain */
                                 low   WORD,  /* high and low   */
                                 high  WORD)  /* order words.   */
                                AT (@time_in_sec),

            data_seg_p          POINTER,      /* Used to obtain
                                                 loc of data seg.*/

            data_seg_p_o        STRUCTURE(    /* Overlay used to */
                                 offset WORD,/* obtain loc of    */
                                 base   WORD)/* data segment.    */
                                 AT (@data_seg_p);
```

```
$subtitle('Time maintenance task')
/************************************************************
*   maintain_time                                           *
*                                                           *
*     This procedure is run as an iRMX 86 task. It repeatedly *
*     performs the following algorithm-                     *
*                                                           *
*           Sleep 1 second.                                 *
*           Gain exclusive access to time_in_sec.           *
*           Add 1 to time_in_sec.                           *
*           Surrender exclusive access to time_in_sec.      *
*                                                           *
*     If the last three steps in the preceding              *
*     algorithm require more than one nucleus time unit,    *
*     the time_in_sec counter will run slow.                *
*                                                           *
*     This procedure must not be called by any procedure    *
*     other than init_time.                                 *
************************************************************/

maintain_time: PROCEDURE REENTRANT;
    DECLARE    status  WORD;

timer_loop:
     DO FOREVER;

        CALL rq$sleep( 100, @status );  /* Sleep for one
                                           second. */

        CALL rq$receive$control         /* Gain exclusive */
                (time_region, @status); /* access.     */

        time_in_sec_o.low =             /* Add 1 second */
                time_in_sec_o.low +1;   /* to low order */
                                        /* half of timer.*/

        IF (time_in_sec_o.low = 0)      /* Handle overflow.*/
            THEN time_in_sec_o.high =
                time_in_sec_o.high + 1;

        CALL rq$send$control(@status);  /* Surrender access*/

     END timer_loop;
END maintain_time;
```

```
$subtitle('Get Time')
/*************************************************************
*   get time                                                 *
*                                                            *
*     This procedure is called by the application code       *
*     in order to obtain the contents of time_in_sec.        *
*     This procedure can be called any number of times.      *
*************************************************************/

get_time: PROCEDURE DWORD REENTRANT PUBLIC;

   DECLARE   time   DWORD,
             status WORD;

   CALL rq$receive$control              /* Gain exclusive */
             ( time_region, @status);   /* access.        */

   time = time_in_sec;

   CALL rq$send$control(@status);       /* Surrender access.*/

   RETURN( time );

END get_time;


$subtitle('Set Time')
/*************************************************************
*   set_time                                                 *
*                                                            *
*     Application code can use this procedure to place       *
*     a specific double word value into time_in_sec.         *
*     This procedure can be called any number of times.      *
*************************************************************/

set_time: PROCEDURE( time ) REENTRANT PUBLIC;

   DECLARE time    DWORD,
           status  WORD;

   CALL rq$receive$control             /* Gain exclusive access.*/
           (time_region, @status);

   time_in_sec = time;                 /* Set new time. */

   CALL rq$send$control(@status);      /* Surrender access. */

END set_time;
```

```
$subtitle('Initialize Time')
/************************************************************
*  init_time                                                *
*                                                           *
*    This procedure zeros the timer, creates a task to      *
*    maintain the timer, and a region to ensure exclusive   *
*    access to the timer.  This procedure must be called    *
*    before the first time that get_time or set_time is     *
*    called.  Also, this procedure should be called only    *
*    once.  The easiest way to make sure this happens is to *
*    call init_time from your initialization task.          *
*                                                           *
*    The timer task will run in the job from which this     *
*    procedure is called.                                   *
*                                                           *
*  If your application experiences a lot of interrupts,     *
*    the timer may run slow.  You can rectify this          *
*    problem by raising the priority of the timer           *
*    task.  To do this, change the 128 in the               *
*    rq$create$task system call to a smaller number.        *
*    This change may slow the processing of your            *
*  interrupts.                                              *
************************************************************/

init_time: PROCEDURE(ret_status_p) REENTRANT PUBLIC;

    DECLARE ret_status_p          POINTER,
            ret_status            BASED ret_status_p WORD,
            timer_task_t          TASK,
            local_status          WORD;

    time_in_sec = 0;

    time_region = rq$create$region          /* Create a region. */
                    (PRIORITY_QUEUE, ret_status_p);

    IF (ret_status <> E$OK) THEN
        RETURN;                             /* Return w/ error. */

    data_seg_p = @data_seg_p;               /* Get contents of
                                               DS register. */

    timer_task_t = rq$create$task           /* Create timer task. */
                    (128,                   /* priority          */
                     @maintain_time,        /* start addr        */
                     data_seg_p_o.base,     /*  data seg base    */
                     0,                     /*  stack ptr        */
                     512,                   /*  stack size       */
                     0,                     /*  task flags       */
                     ret_status_p);
```

```
    IF (ret_status <> E$OK) THEN
       CALL rq$delete$region                  /* Since could not */
            (time_region, @local_status);     /* create task,    */
                                              /* must delete     */
                                              /* region.         */

END init_time;

END timer;
```

BIBLIOGRAPHY

The following reading material contains information that relates to interface procedures, INCLUDE files, the PL/M-86 language, iRMX 86 configuration, iRMX 86 tasks, and the LINK86 command:

- Chapter 3 of iRMX 86™ PROGRAMMING TECHNIQUES

  This chapter provides a discussion of iRMX 86 interface procedures.

- ISIS-II PL/M-86 COMPILER OPERATOR'S MANUAL
  Order Number 9800478

  This manual discusses the concept of using INCLUDE files to add source statements to the file currently being compiled.

- PL/M-86 PROGRAMMING MANUAL
  Order Number 9800466

  This manual describes the PL/M-86 language.

- iRMX 86™ CONFIGURATION GUIDE FOR ISIS-II USERS
  Order Number 9803126

  This manual describes the process of configuring an application system that uses the iRMX 86 Operating System. Included in this description is a discussion of initialization tasks.

- iRMX 86™ NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL
  Order Number 9803122

  This manual thoroughly describes the concept of an iRMX 86 task.

- MCS-86 SOFTWARE DEVELOPMENT UTILITIES OPERATING INSTRUCTIONS FOR ISIS-II USERS
  Order Number 9800639

  This manual describes the LINK86 command which you must use to link this timer module to your application software.

- iRMX 86™ SYSTEM PROGRAMMER'S REFERENCE MANUAL
  Order Number 142721

  This manual explains the use of iRMX 86 regions.

CHAPTER 6. CALLING THE iRMX 86™ SYSTEM FROM ASSEMBLY LANGUAGE

This chapter is for anyone who wants to use iRMX 86 system calls from programs written in MCS-86 assembly language. In order to be able to use system calls from assembly language, you should be familiar with the following concepts:

- iRMX 86 system calls
- iRMX 86 interface procedures
- PL/M-86 size controls

You should also be familiar with PL/M-86 and fluent in MCS-86 assembly language (ASM86). If you are unfamiliar with any of this information, refer to the bibliography at the end of this chapter for additional reading.

PURPOSE OF THIS CHAPTER

The purpose of this chapter is twofold. First, it briefly outlines the process involved in using an iRMX 86 system call from an assembly language program. Second, it directs you to other INTEL manuals that provide either background information or details concerning interlanguage procedure calls.

CALLING THE SYSTEM

If you read the "Interface Procedures and Libraries" chapter of this manual, you found that your programs communicate with the iRMX 86 System by calling interface procedures that are designed for use with programs written in PL/M-86. So the problem of using system calls from assembly language programs becomes the problem of making your assembly language programs obey the procedure-calling protocol used by PL/M-86. For example, if your ASM86 programs uses the SEND$MESSAGE system call, then you must call rq$send$message interface procedure from your assembly language code.

NOTE

The techniques for calling PL/M-86 procedures from assembly language are completely described in the ISIS-II PL/M-86 COMPILER OPERATOR'S MANUAL and the MCS-86 ASSEMBLER OPERATING INSTRUCTIONS FOR ISIS-II USERS.

SELECTING A SIZE CONTROL

Before you begin writing your assembly language calls to the PL/M-86 interface procedures, you should be aware that the conventions used to communicate between the two languages depend upon the size control (COMPACT, MEDIUM, or LARGE) of the interface procedures you use. Consequently, you must select a size control before you write your interlanguage procedure calls.

If all of your application is written in assembly language, you can arbitrarily select a size control and use the libraries for the selected control. However, you can obtain a size and performance advantage by using the COMPACT interface procedures, since their procedure calls are all NEAR. The LARGE interface, which has procedures that require FAR procedure calls, is only advantageous if your application code is larger than 64K bytes.

On the other hand, if some of your application code is written in PL/M-86, your assembly language code should use the same interface procedures as are used by your PL/M-86 code.

BIBLIOGRAPHY

The following reading material contains information that relates to iRMX 86 interface procedures, iRMX 86 system calls, the PL/M-86 language, the MCS-86 assembly language, and techniques for calling procedures of one language from the other:

- Chapter 3 of iRMX 86™ PROGRAMMING TECHNIQUES

  This chapter discusses iRMX 86 interface procedures.

- iRMX 86™ NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL
  Order Number 9803122

  This manual provides the names of many iRMX 86 system calls as well as a description of each of the required parameters and the order in which the parameters must appear.

- iRMX 86TM I/O SYSTEM AND LOADER REFERENCE MANUAL
  Order Number 9803123

  This manual provides the names of many iRMX 86 system calls as well as a description of each of the required parameters and the order in which the paramters must appear.

- iRMX 86TM SYSTEM PROGRAMMER'S REFERENCE MANUAL
  Order Number 142721

  This manual provides the names of some iRMX 86 system calls as well as a description of each of the required parameters and the order in which they must appear.

- ISIS-II PL/M-86 COMPILER OPERATOR'S MANUAL
  Order Number 9800478

  This manual contains a discussion of procedure-calling conventions used by the PL/M-86 compiler. These are the conventions that your assembly language program must follow when it uses iRMX 86 system calls. The manual also discusses PL/M-86 size controls.

- PL/M-86 PROGRAMMING MANUAL
  Order Number 9800466

  This manual describes the PL/M-86 language.

- MCS-86 ASSEMBLER OPERATING INSTRUCTIONS FOR ISIS-II USERS
  Order Number 9800641

  This manual explains how modules written in ASM86 can communicate with modules written in PL/M-86.

- MCS-86 ASSEMBLY LANGUAGE REFERENCE MANUAL
  Order Number 9800640

  This manual describes the MCS-86 asssembly language.

CHAPTER 7. COMMUNICATION BETWEEN iRMX^{TM} 86 JOBS

This chapter applies to anyone who wants to pass information from one iRMX 86 job to another. In order to understand this chapter, you must be familiar with the following concepts:

- iRMX 86 jobs, including object directories
- iRMX 86 tasks
- iRMX 86 segments
- the root job of an iRMX 86-based system
- iRMX 86 mailboxes
- iRMX 86 physical files or named files
- iRMX 86 stream files
- iRMX 86 type managers and composite objects

If you are unfamiliar with any of these concepts, refer to the bibliography at the end of this chapter for additional reading.

PURPOSE OF THIS CHAPTER

In multiprogramming systems, where each of several applications is implemented as a distinct iRMX 86 job, there is an occasional need to pass information from one job to another. This chapter describes several techniques that you can use to accomplish this.

The techniques are divided into two collections. The first collection deals with passing large amounts of information from one job to another, while the second collection deals with passing iRMX 86 objects.

PASSING LARGE AMOUNTS OF INFORMATION BETWEEN JOBS

There are three methods for sending large amounts of information from one job to another:

1) You can create an iRMX 86 segment and place the information in the segment. Then, using one of the techniques discussed below for passing objects between jobs, you can deliver the segment.

The advantages of this technique are:

- Since this technique requires only the Nucleus, you can use it in systems that do not use other iRMX 86 subsystems.

- The iRMX 86 Operating System does not copy the information from one place to another.

The disadvantages of this technique are:

- The segment will occupy memory until it is deleted, either explicitly (by means of the DELETE$SEGMENT system call), or implicitly (when the job that created the segment is deleted). Until the segment is deleted, a substantial amount of memory is unavailable for use elsewhere in the system.

- The application code may have to copy the information into the segment.

2) You can use an iRMX 86 I/O System stream file.

The advantages of this technique are:

- The data need not be broken into records.

- This technique can easily be changed to Technique 3.

The disadvantage of this technique is that you must configure the Basic I/O System into your application system.

3) You can use the iRMX 86 I/O System to write the information onto a mass storage device, from which the job needing the information can read it.

The advantages of this technique are:

- Many jobs can read the information.

- This technique can easily be changed to Technique 2.

- The information need not be divided into records.

The disadvantages of this technique are:

- You must incorporate the iRMX 86 Basic I/O System into your application system.

- Device I/O is slower than reading and writing to a stream file.

PASSING OBJECTS BETWEEN JOBS

Jobs can also communicate with each other by sending objects across job boundaries. You can use any of several techniques to accomplish this, and you should avoid using one seemingly straightforward technique. In the following discussions you will see how to pass objects by using object directories, mailboxes, and parameter objects. You will also see why you should not pass object tokens by embedding them in an iRMX 86 segment or in a fixed memory location.

PASSING OBJECTS THROUGH OBJECT DIRECTORIES

For the purpose of this discussion, consider a hypothetical system in which tasks in separate jobs must communicate with each other. Specifically, suppose that Task B in Job B must not begin or resume running until Task A in Job A grants permission.

One way to perform this synchronization is to use a semaphore. Task B can repeatedly wait at the semaphore until it receives a unit, and Task A can send a unit to the semaphore whenever it wishes to grant permission for Task B to run. If Tasks A and B are within the same job, this would be a straightforward use of a semaphore. But the two tasks are in different jobs, and this causes some complications.

Specifically, how do Tasks A and B access the same semaphore? For instance, Task A can create the semaphore and access it, but how can Task A provide Task B with a token for the semaphore? The trick is to use the object directory of the root job.

In the following explanation, each of the two tasks must perform half of a protocol. The process of creating and cataloging the semaphore is one half, and the process of looking up the semaphore is the other.

In order for this protocol to succeed, the programmers of the two tasks must agree on a name for the semaphore, and they must agree which task performs which half of the protocol. In this example, the semaphore is named permit_sem. And, because Task B must wait until Task A grants permission, Task A will create and catalog the semaphore, and Task B will look it up.

Task A performs the creating and cataloging as follows:

1) Task A creates a semaphore with no units by calling the CREATE$SEMAPHORE system call. This provides Task A with a token for the semaphore.

2) Task A calls the GET$TASK$TOKENS system call to obtain a token for the root job.

3) Task A calls the CATALOG$OBJECT system call to place a token for the semaphore in the object directory of the root job under the name permit_sem.

4) Task A continues processing, eventually becomes ready to grant permission, and sends a unit to permit_sem.

Task B performs the look-up protocol as follows:

1) Task B calls the GET$TASK$TOKENS system call to obtain a token for the root job.

2) Task B calls the LOOKUP$OBJECT system call to obtain a token for the object named permit_sem. If the name has not yet been cataloged, Task B waits until it is.

3) Task B calls the RECEIVE$UNITS system call to request a unit from the semaphore. If the unit is not available then Task A has not yet granted permission, and Task B waits. When a unit is available, Task A has granted permission, and Task B becomes ready.

There are several aspects of this technique that you should be aware of:

- In the example, the object directory technique was used to pass a semaphore. The same technique can be used to pass any type of iRMX 86 object.

- The semaphore was passed via the object directory of the root job. The root job's object directory is unique in that it is the only object directory to which all jobs in the system can gain access. This accessibility allows one job to "broadcast" an object to any job that knows the name under which the object is cataloged.

- The object directory of the root job must be large enough to accommodate the names of all the objects passed in this manner. If it is not, it will become full and the iRMX 86 Operating System will return an exception code when attempts are made to catalog additional objects.

- If you use this technique to pass many objects, you could have problems ensuring unique names. If name management becomes a problem, different sets of jobs can adopt the convention of using an object directory other than that of the root job. To accomplish this, one of the jobs catalogs itself in the root job's object directory under an agreed-upon name. The other jobs can then look up the cataloged job and use its object directory rather than that of the root job.
- In the example, the object-passing protocol was divided into two halves--the create-and-catalog half, and the look-up half. The protocol works correctly regardless of which half starts to run first.

PASSING OBJECTS THROUGH MAILBOXES

Another means of sending objects from one job to another is to use a mailbox. This is a two-step process in that the two jobs using the mailbox must first use the object directory technique to obtain mutual access to the mailbox, and then they use the mailbox to pass additional objects.

PASSING PARAMETER OBJECTS

One of the parameters of the CREATE$JOB system call is a parameter object. The purpose of this parameter is to allow a task in the parent job to pass an object to the newly created job. Once the tasks in the new job begin running, they can obtain a token for the parameter object by calling GET$TASK$TOKENS. This technique is illustrated in the following example:

Suppose that Task 1 in Job 1 is responsible for spawning a new job (Job 2). Suppose also that Task 1 maintains an array that is needed by Job 2. Task 1 can pass the array to Job 2 by putting the array into an iRMX 86 segment, and designating the segment as the parameter object in the CREATE$JOB system call. Then the tasks of Job 2 can call the GET$TASK$TOKENS system call to obtain a token for the segment.

In the foregoing example, the parameter object is a segment. However, you can use this technique to pass any kind of iRMX 86 object.

AVOID PASSING OBJECTS THROUGH SEGMENTS OR FIXED MEMORY LOCATIONS

In the current version of the iRMX 86 Operating System, tokens remain unchanged when objects are passed from job to job. However, Intel reserves the right to modify this rule. In other words, if you pass objects from one job to another and you want your software to be able to run on future releases of the iRMX 86 System, obey the following guidelines:

- Never pass a token from one job to another by placing the token in an iRMX 86 segment and then passing the segment.
- Never pass a token from one job to another by placing the token in any memory location that the two jobs both access.

COMPARISION OF OBJECT-PASSING TECHNIQUES

There are serveral guidelines to consider when deciding how to pass an object between jobs:

- If you are passing only one object from a parent job to a child job, use the parameter object when the parent creates the child.
- If you are passing only one object but not from parent to child, use the object directory technique. It is simpler than using a mailbox.
- If you need to pass more than one object at a time, you can use any of the following techniques:
  - Assign an order to the objects and send them to a mailbox where the receiving job can pick them up.
  - Give each of the objects a name and use an object directory.
  - Write a simple type manager that packs and unpacks a set of objects. Then pass the set of objects as one composite object.

BIBLIOGRAPHY

The following reading material contains information that relates to iRMX 86 jobs, tasks, segments, mailboxes, files, type managers, and composite objects:

- iRMX 86™ NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL
  Order Number 9803122

  This manual describes iRMX 86 jobs, tasks, segments, and mailboxes as well as the system calls that manipulate them.

- iRMX 86™ I/O SYSTEM AND LOADER REFERENCE MANUAL
  Order Number 9803123

  This manual describes iRMX 86 stream, physical, and named files.

- iRMX 86™ CONFIGURATION GUIDE FOR ISIS-II USERS
  Order Number 9803126

  This manual describes the iRMX 86 root job.

- iRMX 86™ SYSTEM PROGRAMMER'S REFERENCE MANUAL
  Order Number 142721

  The manual describes type managers and composite objects.

CHAPTER 8. SIMPLIFYING CONFIGURATION DURING DEVELOPMENT

This chapter is for anyone who writes procedures that run as initial tasks during the system initialization process. In order to understand this chapter, you should be familiar with the following information:

- the iRMX 86 configuration process
- the use of LINK86
- the use of LOC86

If you are unfamiliar with any of these concepts, refer to the bibliography at the end of this chapter for additional reading.

PURPOSE OF THIS CHAPTER

While you are creating your application jobs, you will probably use the following iterative procedure to remove bugs from your code:

1) Configure your system.

2) Test the system to find bugs.

3) If any bugs are found, modify the application code to eliminate the bugs and go to Step 1.

In order to remove most of the bugs from your application software, you might have to loop several times through these three steps. Consequently, you may spend a substantial amount of effort configuring your system.

The purpose of this chapter is to show you how to simplify the process of configuring your system during development. By using the techniques presented here, you can reduce the time you spend in configuration and increase the time available for debugging.

SIMPLIFYING CONFIGURATION DURING DEVELOPMENT

SUMMARY OF CONFIGURATION

Configuration is a four-phase process:

1) Select the iRMX 86 software that meets the needs of your application.

2) Decide where in memory to place the modules of code and the data segments.

3) Link and locate the code and data.

4) Tell the root job where the code and data are located.

Once you have performed these four phases, you need only load the code and start up the root job in order to get the entire system running.

CONFIGURATION AND DEBUGGING

During the process of debugging an application, you generally perform Phase 1 of configuration only once, and Phases 2 through 4 repeatedly. You need not repeat Phase 1 because your application generally uses the same set of iRMX 86 system calls throughout debugging. On the other hand, Phases 2 through 4 are generally repeated because the application software modules change frequently during debugging.

By using a special method during the coding of your initial task software, you can freeze the locations of your application software modules and data segments. This reduces the probability of your repeating Phases 2 and 4 of the configuration process.

THE TECHNIQUE

The %JOB macro used during the configuration process requires three parameters that are very volatile during development. These parameters are exception_handler_entry, init_task_entry, and data_segment_base.

During debugging, as you modify code and (consequently) change the size of your code modules, the values that you must assign to these three parameters are very likely to change. By heeding the following two suggestions, you can significantly reduce the likelihood of changing these parameters and, hence, you can retest your revised application job after merely linking and loading.

FREEZING THE BASE OF THE DATA SEGMENT

If, during development, you locate your job's data segment after your job's code segment, you can freeze the base of the data segment by padding the code segment. Consider the following two situations.

In Job A (Figure 8-1), the code modules are located contiguously, with the data segment immediately following the last module. If any of the modules in Job A grow or shrink as a result of debugging, you must relocate the data segment. This involves changing the data_segment_base parameter of the %JOB macro for the job and regenerating the root job.

In contrast, Job B (Figure 8-1) is designed to accommodate modification. The modules are still located contiguously, but some unused memory has been left between the code segment and the data segment. This unused memory, called padding, allows the modules in the code segment to grow without causing a change in the base address of the data segment.

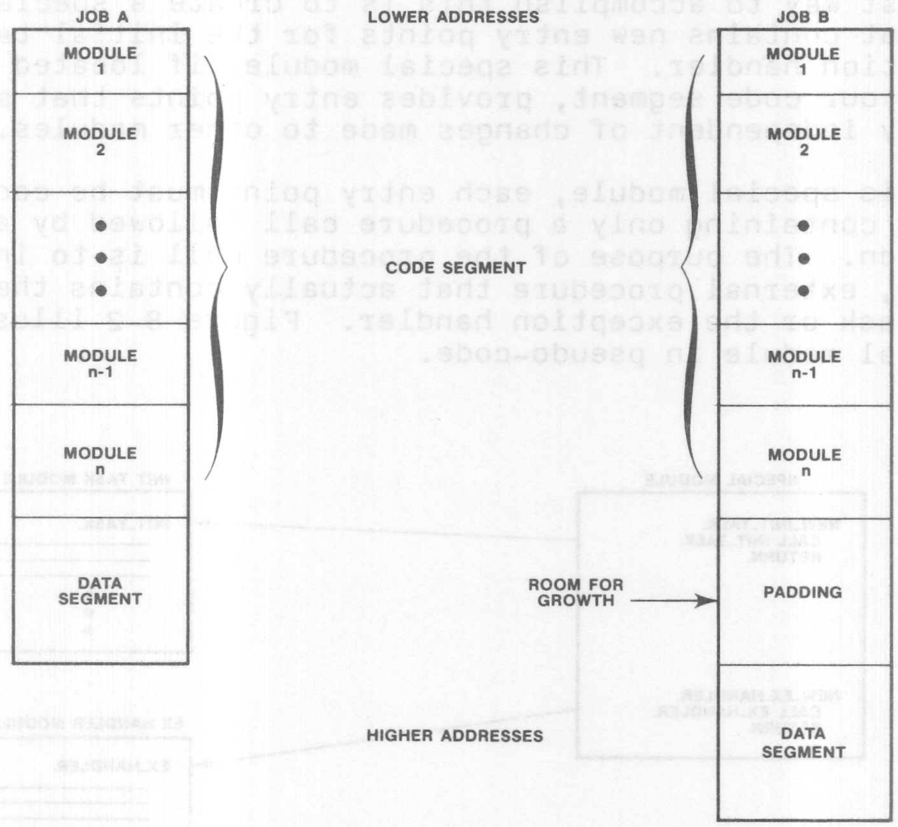

Figure 8-1. How to freeze the base of the data segment

You must decide how much padding to leave between the code and data segments. In general, the less stable the code is, the more padding you should leave. If you are uncertain, try starting with 1000 bytes.

In order to leave the padding between the code and data segments you can use the address control of the LOC86 command. For example,

    ADDRESSES(CLASSES(CODE(aaaaa), DATA(bbbbb)))

where aaaaa is the address at which you want to place the job's code segment, and bbbbb is the address at which you want to place the job's data segment. You can compute bbbbb by adding the size of the padding to the address of the end of the code segment.

FREEZING THE ENTRY POINTS

The %JOB macro requires the addresses of two entry points, one for the job's initial task, and one for the job's exception handler. Because these addresses are expressed as offsets from the base of the job's code segment, you can freeze the addresses by preventing the offsets from changing.

The easiest way to accomplish this is to create a special module that contains new entry points for the initial task and the exception handler. This special module, if located at the front of your code segment, provides entry points that are completely independent of changes made to other modules.

Within this special module, each entry point must be coded as a procedure containing only a procedure call followed by a return instruction. The purpose of the procedure call is to invoke a secondary, external procedure that actually contains the initial task or the exception handler. Figure 8-2 illustrates the special module in pseudo-code.

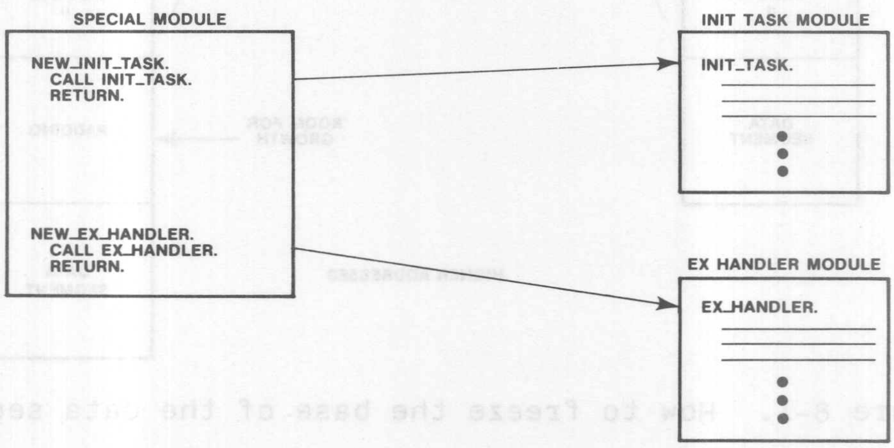

Figure 8-2. Special module freezes entry points

You can place the special module at the front of your code segment (Figure 8-3) by linking it first during the linking process. This will ensure that the new entry points for the initial task and the exception handler are ahead of the code modules that are subject to change. This, in turn, ensures that the new entry points will remain a fixed distance from the base of the code segment, and that you will not need to modify the exception_handler_entry or the init_task_entry parameters.

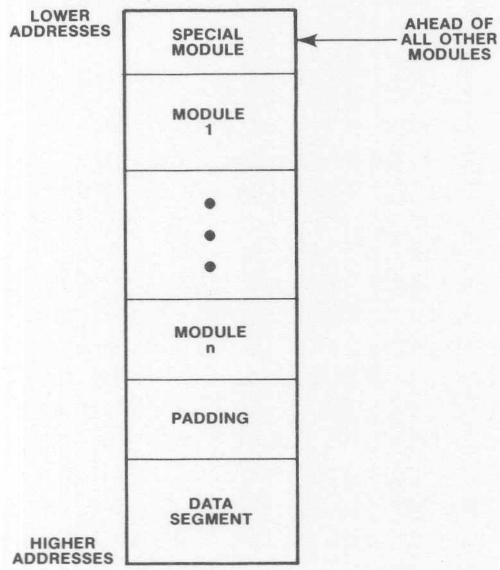

Figure 8-3. Location of the special module

BIBLIOGRAPHY

The following reading material contains information that relates to configuration of iRMX 86-based systems and to the LINK86 and LOC86 commands:

- iRMX 86™ CONFIGURATION GUIDE FOR ISIS-II USERS
  Order Number 9803126

  This manual provides a detailed discussion of the process of configuring an iRMX 86-based system. This discussion includes definitions of initial task and the root job, as well as an explanation of the %JOB macro. The manual also gives explicit directions for deciding where to place the root job in memory.

- MCS-86 SOFTWARE DEVELOPMENT UTILITIES OPERATING INSTRUCTIONS FOR ISIS-II USERS
  Order Number 9800639

  This manual discusses the LINK86 and LOC86 commands.

CHAPTER 9. DEADLOCK AND DYNAMIC MEMORY ALLOCATION

This chapter is for anyone who writes tasks which dynamically allocate memory, send messages, create objects, or delete objects. In order to understand this chapter, you should be familiar with the following concepts:

- memory management in the iRMX 86 Operating System
- using either iRMX 86 semaphores or regions to obtain mutual exclusion

If you are unfamiliar with any of these concepts, refer to the bibliography at the end of this chapter for additional reading.

PURPOSE OF THIS CHAPTER

Memory deadlock is not difficult to diagnose or correct, but it is difficult to detect. Because memory deadlock generally occurs under unusual circumstances, it can lie dormant throughout development and testing, only to bite you when your back is turned. The purpose of this chapter is to provide you with some special techniques that can prevent memory deadlock.

HOW MEMORY ALLOCATION CAUSES DEADLOCK

The following example illustrates the concept of memory deadlock and shows the danger that iRMX 86 tasks can face when they cause memory to be allocated dynamically.

Suppose that the following circumstances exist for Task A and B which belong to the same job:

- Task A has lower priority than Task B.
- Each task wants two iRMX 86 segments of a given size, and each asks for the segments by calling the CREATE$SEGMENT system call repeatedly until both segments are acquired.
- The job's memory pool contains only enough memory to satisfy two of the requests.
- Task B is asleep and Task A is running.

Now suppose that the following events occur in the order listed:

1) Task A gets its first segment.

2) An interrupt occurs and Task B is awakened. Since Task B is of higher priority than Task A, Task B becomes the running task.

3) Task B gets its first segment.

The two tasks are now deadlocked. Task B remains running and continues to ask for its second segment. Not only are both of the tasks unable to progress, but Task B is consuming a great deal, perhaps all, of the processor time. At best, the system is seriously degraded.

This kind of memory allocation deadlock problem is particularly insidious because it quite likely would not occur during debugging. The reason for this is that the order of events is critical in this deadlock situation.

Note that the key event in the deadlock example is the awakening of Task B just after Task A invokes the first CREATE$SEGMENT system call, but just before Task A invokes the second CREATE$SEGMENT call. Because this critical sequence of events occurs only rarely, a "thoroughly debugged" system might, after a period of flawless performance, suddenly fail.

Such intermittent failures are costly to deal with once your product is in the field. Consequently, the most economical method for dealing with memory deadlock is to prevent it.

SYSTEM CALLS THAT CAN LEAD TO DEADLOCK

A task cannot cause memory deadlock unless it causes memory to be allocated dynamically. And the only means for a task to allocate memory is by using system calls. If your task uses any of the following system calls, you must take care to prevent deadlock:

- any system call that creates an object
- any system call belonging to a subsystem other than the Nucleus
- SEND$MESSAGE
- DELETE$JOB
- DELETE$EXTENSION

If a task uses none of the preceding system calls, it cannot deadlock as a result of memory allocation.

PREVENTING MEMORY DEADLOCK

Using any one of the following techniques, you can eliminate memory deadlock from your system:

- When a task receives an E$MEM condition code, the task should not endlessly repeat the system call that led to the code. Rather, it should repeat the call only a predetermined number of times. If the task still receives the E$MEM condition, it should delete all its unused objects, and try again. If the E$MEM code is still received, the task should sleep for a while and then reissue the system call.

- If you have designed your system so that a job cannot borrow memory from the pool of its parent, you can use an iRMX 86 semaphore or region to govern access to the memory pool. Then, when a task requires memory, it must first gain exclusive access to the job's memory pool. Only after obtaining this access may the task issue any of the system calls listed above.

  The task's behavior should then depend upon whether the system can satisfy all of the task's memory requirements:

  - If the system cannot satisfy all requirements, the task should delete any objects that were created and surrender the exclusive access. Then the task should again request exclusive access to the pool.

  - If, on the other hand, all requests are satisfied, the task should surrender exclusive access and begin using the objects.

  This technique prevents deadlock by returning unused memory to the memory pool, where it may be used by another task.

- If you have designed your system so that a job cannot borrow memory from the pool of its parent, prevent the tasks within the job from directly completing for the memory in the job's pool. You can do this by allowing no more than one task in each job to use the system calls listed earlier.

BIBLIOGRAPHY

The following reading material contains information that relates to iRMX 86 memory management, and the use of regions and semaphores.

- iRMX 86™ NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL
  Order Number 9803122

  This manual contains a discussion of memory management in the iRMX 86 Operating System. It also provides detailed information regarding the use of semaphores.

- iRMX 86™ SYSTEM PROGRAMMER'S REFERENCE MANUAL
  Order Number 142721

  This manual discusses iRMX 86 regions and mutual exclusion.

- INTRODUCTION TO THE iRMX 86™ OPERATING SYSTEM
  Order Number 9803124

  This manual explains how you can use a semaphore to obtain mutual exclusion.

CHAPTER 10. PROCEDURES FOR I/O USING A TERMINAL

This chapter is for anyone who creates programs which read (or write) numbers or character strings from (or to) a terminal. In order to use the information in this chapter, you must be familiar with one of the following techniques for communicating with a terminal:

- the iRMX 86 Terminal Handler
- the 957A CO and CI procedures
- the 957A CO procedure in combination with the 957A READ procedure
- sending characters to and getting characters from the terminal's I/O port

If you are not familiar with any of these techniques, refer to the manuals listed in the bibliography at the end of this chapter.

PURPOSE OF THIS CHAPTER

This chapter outlines a strategy for writing a collection of procedures that simplify input from and output to a terminal. Rather than providing the source code, this chapter only describes the function of each procedure.

You can use these descriptions to write procedures that communicate with terminals attached directly to your application hardware or to your Intellec microcomputer development system. Furthermore, you can use these procedures in conjunction with any of the terminal I/O techniques listed earlier in this chapter.

OVERVIEW

Sometimes, either within your application or while you are debugging it, you will find a need to read or write numbers and character strings from a terminal. For instance, suppose that while you are debugging you need to display the contents of a WORD each time a specific mailbox is used. If you attempt to use one of the basic terminal I/O techniques (listed earlier) to do this, you must first convert the number to ASCII-encoded decimal or hexadecimal, and you must send it to the terminal.

The situation is even more complex for input. If you want to read a number from the terminal, you must scan the line to find the beginning of the number, and then you must convert from the ASCII representation to binary.

The following procedures eliminate much of this difficulty. Built upon the basic terminal I/O techniques listed earlier in this chapter, these procedures allow you, with very few lines of code, to read and write individual characters, hexadecimal numbers, decimal numbers, and strings of characters. Furthermore, when these procedures read or write numbers, they return the value of the number, rather than the ASCII representation.

ELEMENTARY PROCEDURES

If you write the following two elementary procedures, you can use them to construct the more advanced terminal I/O procedures:

- GET_CHAR is a procedure that reads a single character from the keyboard and returns a BYTE containing the ASCII representation of the character to the calling procedure. The form of the call in PL/M-86 is

  ```
  value = GET_CHAR;
  ```

  When you write this procedure, you should consider echoing the character to the terminal. If the character is not echoed, the operator may become confused.

- PUT_CHAR is a procedure that requires a BYTE as an input parameter. It displays the contents of the BYTE as an ASCII character on the terminal. The call has the following form in PL/M-86:

  ```
  CALL PUT_CHAR( value );
  ```

These two procedures are important because they provide your more advanced procedures with some degree of device independence. Regardless of which basic terminal I/O technique you use to implement GET_CHAR and PUT_CHAR, you can use the same collection of advanced procedures.

ADVANCED PROCEDURES

Once you have programmed GET_CHAR and PUT_CHAR, you can build a collection of more powerful procedures that support terminal I/O for numbers and character strings.

PROCEDURES FOR I/O USING A TERMINAL

PROCEDURES FOR NUMBERS

There are four useful procedures for reading and displaying numbers via a terminal. Two of the procedures are for hexadecimal values, and two are for decimal.

Procedures for Hexadecimal Numbers

The following procedures are for reading and displaying hexadecimal numbers.

- GET_HEX is a parameterless procedure that reads hexadecimal digits from the keyboard and places the corresponding binary value in a WORD. The form of the call in PL/M-86 is

  ```
  value = GET_HEX;
  ```

  This procedure should use GET_CHAR to skip characters until a hexadecimal character (0-9, A-F, or a-f) is found. Then it should read until one of the following conditions is met:

  - Four hexadecimal characters have been read.
  - A nonhexadecimal character has been read.

  GET_HEX should then convert the hexadecimal characters from ASCII-encoded hexadecimal to binary and return to the calling procedure.

- PUT_HEX requires a WORD as an input parameter and returns no output value. It displays on the terminal the contents of the WORD in hexadecimal. The form of the call in PL/M-86 is

  ```
  CALL PUT_HEX ( value );
  ```

  PUT_HEX should convert the contents of the WORD to four ASCII-encoded hexadecimal characters and should use PUT_CHAR to display the characters on the terminal.

Procedures for Decimal Numbers

The following procedures are for reading and displaying decimal numbers.

- GET_DEC is a parameterless procedure that reads a decimal number from the keyboard and places the corresponding value in a WORD. The form of the PL/M-86 call is

  ```
  value = GET_DEC;
  ```

  This procedure should use GET_CHAR to skip all characters other than digits. Once GET_DEC finds a digit, it should read until one of the following conditions is met:

  - Five digits have been read.
  - A character other than a digit has been read.

  GET_DEC should then convert the digits to a 16-bit binary number and return to the caller. Since 65535 is the largest decimal number that fits into a 16-bit WORD, GET_DEC should return this value whenever it reads a number greater than 65535.

- PUT_DEC requires a WORD as an input parameter, returns no output value, and displays the contents of the WORD on the terminal in decimal. The form of the call in PL/M-86 is

  ```
  CALL PUT_DEC( value );
  ```

  This procedure must first convert the contents of the WORD to ASCII-encoded decimal and then must use PUT_CHAR to display the results.

PROCEDURES FOR STRINGS

There are three procedures for reading and displaying strings. Two are for strings in the form required by the iRMX 86 Operating System, and one is for null-terminated strings.

Procedures for iRMX 86 Strings

The two procedures for reading and displaying iRMX 86 strings are:

- GET_STR, which requires a POINTER as an input parameter, and reads characters from the keyboard until a carriage return is detected. The procedure then places a character count into the byte indicated by the POINTER, and places the characters (in ASCII, one per byte) in the bytes immediately following the count. The form of the call in PL/M-86 is

```
CALL GET_STR( @string );
```

- PUT_STR, which requires a POINTER as an input parameter, displays the iRMX 86 string indicated by the POINTER. The string must consist of a byte (containing a character count) followed by the characters to be displayed. The form of the call in PL/M-86 is

```
CALL PUT_STR( @string );
```

A Procedure for Null-Terminated Strings

Because iRMX 86 strings require a character count, they can be cumbersome for use with large quantities of text. If you foresee a need for displaying large amounts of text, you can avoid manually counting characters by writing a procedure to display a null-terminated string.

The PUT_NT_STR procedure displays a string that, instead of being preceded by a count, is terminated by a byte containing zero (the ASCII null character). The procedure requires, as an input parameter, a pointer that indicates the first character of the string to be displayed. The form of the call in PL/M-86 is

```
CALL PUT_NT_STR( @string );
```

Since you need not count the number of characters read from the terminal, you probably do not need a GET_NT_STR procedure.

BIBLIOGRAPHY

The following manuals contain information about the basic terminal I/O techniques listed earlier in this chapter:

- iRMX 86™ NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL
  Order Number 9803122

  This manual explains how to use the iRMX 86 Terminal Handler to communicate with a terminal attached to your application system.

- iSBC 957A INTELLEC - iSBC 86/12A INTERFACE AND EXECUTION PACKAGE USER'S GUIDE
  Order Number 142849

  This manual explains how to use the 957A CO and CI procedures to communicate with your development system terminal under the control of your application system software. It also explains how to use the READ procedure to read edited lines from the terminal.

CHAPTER 11. GUIDELINES FOR STACK SIZES

This chapter is for anyone who writes tasks that create jobs or tasks. In order to understand this chapter, you must be familiar with the iRMX 86 Debugger, and you must know which system calls are provided by the various subsystems of the iRMX 86 Operating System. You also must know the difference between maskable and nonmaskable interrupts. The bibliography at the end of this chapter lists the documents in which you can find this information.

PURPOSE OF THIS CHAPTER

Whenever you invoke a system call to create a task, you must specify the size of the task's stack. And, since every new job has an initial task that is created simultaneously with the job, you must also designate a stack size whenever you create a job.

When you specify a task's stack size, you should do so carefully. If you specify a number that is too small, your task might overflow its stack and write over information following the stack. This situation can easily cause your system to fail. On the other hand, if you specify a number that is too large, the excess memory will be wasted. So ideally, you should specify a stack size that is equal to or slightly larger than what is actually required.

This chapter provides you with two techniques for estimating the size of your task's stack. One technique is arithmetic, and the other is empirical. For best results, you should start with the arithmetic technique and then use the empirical technique for tuning your original estimate.

ARITHMETIC TECHNIQUE

This technique provides you with a reasonable overestimate of your task's stack size. After you use this technique to obtain a first approximation, you may be able to save several hundred bytes of memory by using the empirical technique described later in this chapter.

The arithmetic technique is based on the fact that there are at most four factors affecting a task's stack. These factors are:

- interrupts.
- borrowing memory across job boundaries.
- iRMX 86 system calls.
- requirements of the task's code. (For example, the stack used to pass parameters to procedures or to hold local variables in reentrant procedures.)

You can estimate the size of a task's stack by summing the amount of memory needed to accommodate these factors. The following sections explain how to compute the stack requirements for the first three factors.

STACK REQUIREMENTS FOR INTERRUPTS

Two considerations govern the amount of stack that a task must reserve to support interrupts: the type of interrupt and the amount of stack that your interrupt handlers use. (Refer to the iRMX 86 NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL for information about interrupt handlers.)

There are two types of interrupts - maskable and nonmaskable. (The distinction between them is described in the iSBC 86/12A HARDWARE REFERENCE MANUAL.)

If your system uses only maskable interrupts, your task's interrupt-related stack requirements are equal to the amount of stack used by your most stack-hungry interrupt handler. This number will always be at least 28 (decimal) bytes.

And, if your system also uses nonmaskable interrupts, you must allow additional stack to accommodate the handler for nonmaskable interrupts. This handler will also require at least 28 (decimal) bytes.

Your task's total interrupt-related stack requirement is the sum of its maskable and nonmaskable requirements.

STACK REQUIREMENTS FOR BORROWING MEMORY

If your task runs in a job that can borrow memory from its parent job, your task might require some stack for memory management. This requirement can affect your task only if your task uses some of the following system calls:

- SEND$MESSAGE or RECEIVE$MESSAGE
- system calls that create or delete objects
- system calls provided by the I/O System or Loader

If your task does use any of these calls and your task's job can borrow memory, you must allow sufficient stack size to accommodate memory management.

Specifically, you must allow 100 (decimal) bytes for each level of borrowing. For example, suppose that your task is in Job C which is a child of Job B which is a child of Job A. If your task uses any of the previously mentioned system calls and Job C can borrow memory from Job B, you must allow 100 bytes of stack. And, if Job B can borrow from Job A, you must allow an additional 100 bytes.

STACK REQUIREMENTS FOR SYSTEM CALLS

When your task invokes an iRMX 86 system call, the processing associated with the call uses some of your task's stack. The amount of stack required depends upon which system calls you use.

Table 11-1 tells you how many bytes of stack your task must have to support various system calls. To find out how much stack you must allocate for system calls, compile a list of all the system calls that your task uses. Scan Table 11-1 to find which of your system calls requires the most stack. By allocating enough stack to satisfy the requirements of the most demanding system call, you can satisfy the requirements of all system calls used by your task.

TABLE 11-1
STACK REQUIREMENTS FOR SYSTEM CALLS

SYSTEM CALLS	BYTES (DECIMAL)
ATTACH$DEVICE ATTACH$FILE CHANGE$ACCESS CREATE$DIRECTORY CREATE$FILE DELETE$FILE DELETE$USER DETACH$DEVICE	600
A$LOAD	450
ANY OTHER I/O SYSTEM CALLS	400
CREATE$JOB DELETE$EXTENSION DELETE$JOB DELETE$TASK FORCE$DELETE RESET$INTERRUPT	375
ANY OTHER NUCLEUS SYSTEM CALLS	200

GUIDELINES FOR STACK SIZES

COMPUTING THE SIZE OF THE ENTIRE STACK

To compute the size of the entire stack, add the following four numbers:

- the number of bytes required for interrupts
- the number of bytes required for memory management
- the number of bytes required for system calls
- the amount of stack required by the task's code segment

You can use the sum of these four numbers as a reasonable estimate of your task's stack requirements. If you desire more accuracy, use the sum as a starting point for the empirical fine tuning described later in this chapter.

EMPIRICAL TECHNIQUE

This technique starts with an overly large stack and uses the iRMX 86 Debugger to determine how much of the stack is unused. Once you have found out how much stack is unused, you can modify your system calls to create smaller stacks.

The cornerstone of this technique is the iRMX 86 Debugger. In order to use the Debugger, you must include it when you configure your application system. Information on how to do this is provided in the iRMX 86 CONFIGURATION GUIDE FOR ISIS-II USERS.

The Inspect Task command of the Debugger provides a display that includes the number of bytes of stack that have not been used since the task was created. If you let your task run a sufficient length of time, you can use the Inspect Task command to find out how much excess memory is allocated to your task's stack. Then you can adjust the stack-size parameter of the system call to reserve less stack.

The only judgment you must exercise when using this technique is deciding how long to let your task run before obtaining your final measurement. If you do not let the task run long enough, it might not encounter the most demanding combination of interrupts, memory borrowing and system calls. This could cause you to underestimate your task's stack requirement and could, consequently, lead to a stack overflow in your final system.

GUIDELINES FOR STACK SIZES

Underestimation of stack size is a risk inherent in this technique. For example, your task might be written so as to use its peak demand for stack only once every two months. Yet you probably don't want to let your system run for two months just to save several hundred bytes of memory. You can avoid such excessive trial runs by padding the results of shorter runs. For instance, you might run your task for 24 hours and then add 200 (decimal) bytes to the maximum stack size. This padding reduces the probability of overflowing your task's stack in your final system.

BIBLIOGRAPHY

The following manuals explain how to use the Debugger, tell which system calls are provided by each subsystem of the iRMX 86 Operating System, and explain the difference between maskable and nonmaskable interrupts.

- iRMX 86 NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL
  Order Number 9803122

  This manual explains how to use the Debugger's Inspect Task command. It also lists most of the system calls provided by the Nucleus.

- iRMX 86 I/O SYSTEM AND LOADER REFERENCE MANUAL
  Order Number 9803123

  This manual lists most of the system calls provided by the I/O System and the Loader.

- iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL
  Order Number 142721

  This manual lists the I/O System and Nucleus system calls not described in the other reference manuals.

- iSBC 86/12A SINGLE BOARD COMPUTER HARDWARE REFERENCE MANUAL
  Order Number 9803074

  This manual explains the difference between maskable and nonmaskable interrupts.

INDEX A. WHEN IS EACH CHAPTER USEFUL?

The purpose of this index is to help you find out which chapters are most useful to you at any given phase of the development of your system.

DIVIDING AN APPLICATION INTO JOBS AND TASKS

WRITING THE CODE FOR TASKS

WHEN IS EACH CHAPTER USEFUL?

<u>DEBUGGING</u>

Chapter 8
Simplifying Configuration During Development

Chapter 10
Procedures for I/O Using a Terminal

<u>CONFIGURATION AND SYSTEM STARTUP</u>

Chapter 8
Simplifying Configuration During Development

INDEX B

For your convenience, the primary reference of each multiple-page topic is <u>underscored</u>.